CS 315-001 – Programming Assignment 1

Jacob Schuetter

28 September 2023

1. **Introduction**

Overall, implementing the various sorting functions was no more difficult than anticipated. I did end up writing more helper functions than I anticipated, most of them specialized for dealing with arrays of pairs, which ended up being the method I used for storing the provided data, in order to avoid losing any information. I did end up changing this method slightly for my implementation of merge sort, but more on that later. One thing that did prove challenging was limiting my functions to the defined elements of the array. In order to avoid having to reallocate arrays, I allocated a number of spaces greater than the largest dataset, which meant, of course, that there were uninitialized elements following the data read into them. I used a slightly different method to deal with this for each implementation (which I’ll detail in future sections), but I did eventually decide to initialize all elements of my arrays with the pair (0,0) before using them.

I decided to implement my sort algorithms to give the results in descending order, as that seemed most appropriate considering the type of data needed. As a result, the runtimes for the “sorted” and “reverse sorted” datasets appear to be flipped. Thus, I will be labeling the “sorted” datasets as “ascending” and “reverse sorted” as “descending” to avoid ambiguity, and any references to data being “sorted” from here on will indicate a sort in descending order.

Ia. Helper Functions

As stated above, most of my helper functions were designed to help me deal with arrays of pairs: reading the data files into them, printing them to the terminal (or later, an output file), or swapping two elements. The first issue I ran into when writing these first functions had to do with determining the best way to use arrays in them. I ended up finding it easier to pass the arrays in as parameters rather than using a return type: a precedent which I carried through to the rest of my function implementations as well. As I mentioned a few lines up, I eventually decided to write the result of my functions to an output .txt file instead of to the terminal, as the terminal history severely limited the available output space. I also found it particularly helpful to write the runtimes of my functions and the corresponding dataset and size to a .csv file so that I could easily generate graphs from it.

1. **Insertion Sort**

Insertion sort proved to be the easiest algorithm to implement (as expected) and used relatively few lines. I did change my approach partway through writing it so that every operation was completed in a single loop rather than using two subsequent loops, but this theoretically had no impact on the runtime of the function. To limit the functionality of the algorithm to assigned elements, I simply used a while loop that would iterate until it found an uninitialized element or reached the maximum number of elements (defined in this case as a constant).

IIa. Runtime

Insertion sort’s runtime complexity analysis showed exactly the types of trends to be expected. The worst case for insertion sort is that in which every element has to be moved to the opposite end of the array one-by-one, hence a reverse-sorted (ascending order) array would be the worst case. The best case is precisely the opposite, in which no element has to be moved at all, i.e. a sorted array. The graph below plotting comparisons v. input size shows the expected results: O(n^2) in the worst case and linear (or nearly so) in the best.

1. **Quicksort**

I took a slightly different approach with quicksort, instead using parameters for the indices of the start and end of the array to limit its functionality, though I obtained these values through a helper function called CountArray that used the same method I did in insertion sort, so at its core, it is still the same thing. It proved the most difficult to write, or at least the most time-consuming, though perhaps only because I forgot to implement a base case in my first version.

IIIa. Runtime

Quicksort’s runtimes were again as expected. In my implementation, I chose the pivot value from the last element in the partition being sorted, so the worst case was, accordingly, the dataset in descending order, as each recursive step was performed on an array of only one fewer element than the last (and an empty array). The dataset in ascending order, rather surprisingly, but still obeys the expected asymptotic behavior of O(n2). As expected, the dataset in random order has the fastest runtime, with a logarithmic trend, since it divides the dataset into (reasonably) equal portions in each recursive step.

1. **Merge Sort**

Rather than using an array of pairs for merge sort as I had the other two algorithms, I decided to use a queue of pairs, as this made reassembling the divided parts significantly easier, given that the only element I cared about was the first element in either division. I did have to implement a much larger if statement than expected in order to handle the cases where one division is emptied before the other has run out of elements. Other than that, implementation was straightforward.

IVa. Runtime

As expected, merge sort had the exact same runtime (relative to input size) on all nine datasets, as displayed by the graph to the right. This makes sense because it follows the same procedure regardless of input: it divides the input into single elements, then recombines them by comparing the first element of two divisions. The trend does seem more linear than the expected logarithmic trend, but given the small range of input values, this is not surprising, as the logarithm function grows quite slowly.

1. **Additional Information**

Generative AI was not used in any part of this project.

Below may be found the table of runtimes for the given datasets.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | n | Insertion sort | Quicksort | Merge sort |
| pokemonRandomLarge.csv | 800 | 157291 | 11436 | 7776 |
| pokemonRandomMedium.csv | 432 | 46938 | 4465 | 3808 |
| pokemonRandomSmall.csv | 166 | 6926 | 1227 | 1238 |
| pokemonReverseSortedLarge.csv | 800 | 3669 | 319600 | 7776 |
| pokemonReverseSortedMedium.csv | 432 | 1197 | 93096 | 3808 |
| pokemonReverseSortedSmall.csv | 166 | 188 | 13695 | 1238 |
| pokemonSortedLarge.csv | 800 | 319600 | 108198 | 7776 |
| pokemonSortedMedium.csv | 432 | 93096 | 36735 | 3808 |
| pokemonSortedSmall.csv | 166 | 13695 | 7902 | 1238 |